SimBench: A Portable Benchmarking Methodology for Full-System Simulators

Harry Wagstaff
University of Edinburgh
Email: hwagstaf@inf.ed.ac.uk

Bruno Bodin
University of Edinburgh
Email: bbodin@inf.ed.ac.uk

Tom Spink
University of Edinburgh
Email: tspink@inf.ed.ac.uk

Björn Franke
University of Edinburgh
Email: bfranke@inf.ed.ac.uk

***Abstract*—Full-system simulators are increasingly finding their way into the consumer space for the purposes of backwards compatibility and hardware emulation (e.g. for games consoles). For such compute-intensive applications simulation performance is paramount. In this paper we argue that existing benchmark suites such as SPEC CPU2006, originally designed for architecture and compiler performance evaluation, are not well suited for the identification of performance bottlenecks in full-system simulators. While their large, complex workloads provide an indication as to the performance of the simulator on 'real-world' workloads, this does not give any indication of why a particular simulator might run an application faster or slower than another. In this paper we present SimBench, an extensive suite of targeted micro-benchmarks designed to run bare-metal on a full-system simulator. SimBench exercises dynamic binary translation (DBT) performance, interrupt and exception handling, memory access performance, I/O and other performance-sensitive areas. SimBench is cross-platform benchmarking framework and can be retargeted to new architectures with minimal effort. For several simulators, including QEMU, Gem5 and SimIt-ARM, and targeting ARM and Intel x86 architectures, we demonstrate that SimBench is capable of accurately pinpointing and explaining real-world performance anomalies, which are largely obfuscated by existing application-oriented benchmarks.**

I. Introduction

Fast instruction set simulation is an increasingly important technology. It is used in both the commercial space (for software development, design space exploration, and debugging), as well as in the consumer space (to provide backwards compatibility). Fast simulation technologies allow us to run large, complex applications which were originally developed for one 'target' architecture (such as ARM) on a machine of a different 'host' architecture (such as x86).

Simulation tools can be broadly split into two groups. 'User-mode' (also sometimes called 'Syscall emulation') simulators permit the execution of a target application on a host machine, in the context of the operating system running on the host machine. The target binary might have access to the host file system, network, and other OS-provided features. On the other hand, 'Full-System' simulation allows the user to run an entire target operating system in a simulated context. This is achieved by modelling each component in the target hardware platform, including the CPU cores, MMU, uncore devices such as timers, I/O devices, etc. This grouping of simulators is independent of if they perform any performance modelling (whether or not they are 'cycle-accurate', e.g. Gem5 [5]), or not (e.g., QEMU [3] and Simics [22]). The requirement for fast full-system simulation has sparked interest in techniques such as Dynamic Binary Translation [33], parallel Just-in-Time compilation [8], precise interrupt handling [10], and fast memory address translation [32].

Since full-system simulation performance is important, it is critical that the speed of these simulators can be accurately evaluated and analysed. The SPEC2006 suite, which is a collection of benchmarks based on real world applications such as the gcc compiler, is used (such as in [28, 31, 26, 32, 9, 15, 7]). This has the advantage of reflecting real-world conditions and applications. However, these programs are large and complex, and may run for many billions of instructions, and so there is only limited insight into the reasons for performance bottlenecks. Other benchmark suites, targeting OS-level operations, or hardware virtualization features (such as LMBench [23] and VMark [30]) also exist, but do not properly exercise full-system simulation techniques. Occasionally (e.g. in [11]), small microbenchmarks might be used for performance analysis. However, this is often an ad-hoc solution, and they may not be available for others to use and compare with. Additionally, a technique which improves the performance of the simulator in one area may unexpectedly degrade the performance of some other operation, which may not be detected if each aspect of the simulator is not thoroughly analysed.

In this paper we demonstrate that neither of these approaches are suitable for the analysis of full-system simulation performance. This is for several reasons: 1) Existing kernel as well as application benchmarks do not sufficiently exercise system-level features, which are efficient in hardware, but costly to simulate, 2) full applications execute many billions of instructions and performance bottlenecks are hidden in complex interactions, hard to isolate and attribute to simulator design choices, 3) an aggregate performance figure further hides individual application behaviour. Instead, in this paper we develop the novel SimBench methodology. SimBench exercises critical contributors to full-system simulation performance using benchmarks targeted specifically at full-system simulation features and technologies. It is easily ported to new ISAs and platforms, is self-contained, and runs as a bootable bare-metal executable. It accurately pinpoints performance deficits, allowing the user to evaluate simulator design and implementation trade-offs and fix performance anomalies.

This paper makes the following contributions:

1) We demonstrate that existing application benchmarks

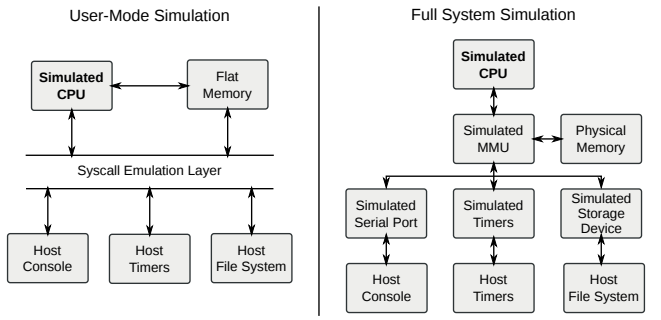


Fig. 1: User-Mode simulation with a simple memory model supporting one virtual address space and syscall emulation vs. Full-System simulation – capable of hosting a full OS – with a simulated MMU and I/O devices backed by host counterparts.

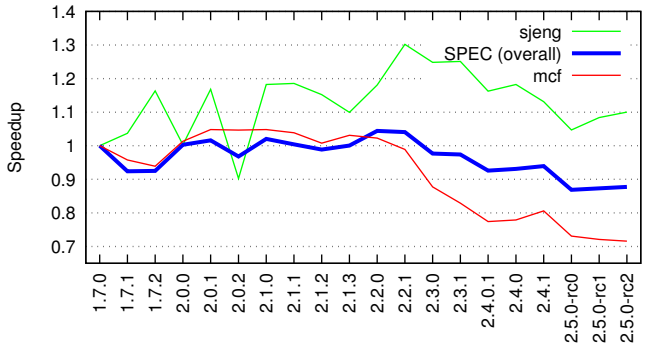


Fig. 2: Relative performance of the `sjeng` and `mcf` benchmarks and the overall SPEC rating (weighted geometric mean across all benchmarks) on a variety of QEMU versions for x86. Version 1.7 is used as a baseline.

such as SPEC CPU2006 are not well suited for the performance analysis of full-system simulators.

2) We develop a retargetable performance evaluation methodology, which exercises specific performance related features in full-system simulators though a set of targeted micro-benchmarks.
3) We show how these detailed performance metrics can be used to drive further simulator development and to model application performance without the need to repeatedly run full-scale application benchmarks.

A. Motivating Example

We performed an experiment into running ARM binaries of the SPEC benchmarks on an ARM Linux OS inside multiple versions of QEMU on an x86 host machine. The Linux kernel and SPEC binaries are all compiled with GCC 5.1 for the ARMv5 architecture, and every version of QEMU has been compiled with the same version of GCC. The results of this experiment are shown in Figure 2.

Compared to version 1.7.0, released in 2013, the current version 2.5.0 has suffered more than 10% performance loss across the SPEC suite. Whilst this is a surprising result in itself, this behaviour is not consistent across all SPEC applications: individual benchmarks have suffered an even greater performance degradation, whilst others exhibit improvements. In fact, we observe a widening performance gap between the `sjeng` and `mcf` applications. The performance of `mcf` has dropped by almost 30% and `sjeng`'s performance has improved by about 10% between the earliest and latest releases. Additionally, `sjeng`'s performance peaked in version 2.2.1, delivering a 30% speedup over version 1.7.0, before gradually suffering losses. These losses are beginning to be addressed in more recent releases, exacerbating the situation for `mcf`.

This motivating example demonstrates that application benchmarks such as the SPEC suite are not effective for performance analysis of full-system simulators. Multiple effects, resulting in speedup for some applications and slowdown for others, can cancel each other out. An average performance profile would not accurately account for the dramatic performance loss of individual applications. Even more important is the fact that the use of application benchmarks does not *explain* the observed behaviour. Developers of the simulator have no indication which change has caused this effect and where performance bottlenecks occur in their system. What is needed is a methodology to benchmark individual performance-critical aspects of full-system simulators, in order to isolate performance anomalies and pinpoint areas for improvement. In Section III-B we will show how our novel SimBench methodology directly identifies control flow and exception handling performance as possible sources of the particular performance regressions shown in Figure 2.

II. SimBench Methodology

SimBench is designed to run bare metal on each target architecture and platform. There is no underlying OS or RTOS (although a bootloader may be used). Each benchmark runs with a configurable iteration count and a per-benchmark run time is reported. The iteration count should be set such that each benchmark runs for a reasonable amount of time (perhaps several minutes), in order to amortise startup and shutdown time. When reporting results, both the run time and iteration counts should be reported.

To ensure that the benchmark runtime consists as much as possible of the 'interesting' operations, each benchmark is executed in three phases. First, benchmark specific setup of page tables, interrupt vectors, etc. is performed. Second, the benchmark kernel itself is executed for the desired number of iterations. Finally, any benchmark specific cleanup is performed. Only the benchmark kernel itself is timed, and so supporting operations do not impact the benchmark runtime.

Finally, it is important that the benchmarks can be strongly optimised, without affecting any behaviours that are being evaluated – we recommend that SimBench is compiled with standard '-O3' optimisations. Several techniques are used to allow the compiler to optimise the benchmark suite where it is desirable, while still guaranteeing correct behaviour. This is done mainly through the use of volatile variables and inline assembly statements. In some cases, control flow is made unpredictable by having it depend on the iteration count, or

Benchmark	Iterations	Operation Density	
		SimBench	SPEC
Code Generation			
Small Blocks	100K	0.049	8.49E-7
Large Blocks	500K	0.003	8.09E-8
Control Flow			
Inter-Page Direct	100M	0.204	0.025
Inter-Page Indirect	250K	0.268	0.022
Intra-Page Direct	500M	0.632	0.131
Intra-Page Indirect	200K	0.286	0.011
Exception Handling			
Data Access Fault	25M	0.143	8.02E-7
Instruction Access Fault	25M	0.063	2.49E-8
Undefined Instruction	50M	0.125	0
System Call	50M	0.100	1.47E-6
External Software Interrupt	20M	† 0.015	† 1.36E-6
I/O			
Memory Mapped Device	400M	† 0.250	† 5.17E-6
Coprocessor Access	250M	0.167	2.83E-4
Memory System			
Cold Memory Access	50M	0.143	9.16E-4
Hot Memory Access	500M	0.909	0.809
Nonprivileged Access	300M	0.125	0
TLB Eviction	4M	0.062	2.51E-6
TLB Flush	4M	0.091	0

Fig. 3: This table shows the set of benchmarks contained within the SimBench suite. Benchmarks marked with a † have significant platform-specific portions which may skew the operation density measurement.

be via function pointers. This adds some overhead to the benchmark but cannot be easily avoided.

A. Features Covered

SimBench currently includes 18 benchmarks in 5 groups. This set of benchmarks has been built after an extensive evaluation of existing full-system simulators, as well as the literature surrounding fast simulation.

We have also decided to omit features which have large platform-specific components, or which are better covered by existing benchmarks. In particular, there is no evaluation of external I/O (which is better covered by I/O benchmarks such as FIO or HDParm), or of translated code quality (which can be covered by the wide range of application benchmarks). We have also avoided the testing of floating point emulation infrastructure, such as rounding mode changes, context save/restore operations etc., although this might be a possible enhancement in future versions. We also avoid 'high-level' benchmarks which test application or OS structures such as algorithmic kernels or file accesses. Other benchmark suites (e.g., LMbench [23]) are better suited for this.

Some of the features measured by SimBench exist on some architectures but not others. For example, the ARM architecture has kernel-mode instructions to access memory without using kernel privileges. There is no equivalent of this in the x86 architecture. SimBench includes a benchmark for this feature, but it is implemented as a no-op for the x86 port.

B. Benchmark Categories

SimBench contains several categories of benchmark outlined in Figure 3. These target code generation performance (for DBT-based simulators), control flow handling, exception/interrupt handling, I/O infrastructure, and memory systems. Figure 3 also provides the default iteration count used for each benchmark (tuned to allow the full benchmark suite to execute in a reasonable amount of time on a variety of platforms and simulators), as well as the 'operation density' of each benchmark. The operation density is the relative number of tested operations performed per executed instruction of the benchmark kernel. For each benchmark, its operation density across the SPEC2006 INT benchmarks is also specified. Some operations do not occur at all during the SPEC benchmarks, such as full TLB flushes, nonprivileged memory accesses, and undefined instruction exceptions. For all operations, the operation density is higher in the SimBench benchmark than it is across the SPEC benchmark suite, showing that SimBench thoroughly exercises the targeted feature in each case.

We now discuss each of these benchmark categories, as well as the individual benchmarks in each category. In particular, we will focus on decisions made when designing these benchmarks and any difficulties and challenges encountered during their design and implementation.

1) Code Generation: The Code Generation benchmarks are mainly designed to measure DBT performance in terms of code generation speed and do not attempt to measure the quality of generated code. In order to measure code generation performance, two separate benchmarks are used: one which contains many small basic blocks, and another which contains one very large basic block. Both of these benchmarks work by executing the same region of code (the many small blocks, or single large block) repeatedly. Between each execution, the code region is rewritten in order to invalidate any DBT translations (or other cached data structures) of the code region. This means that these benchmarks also measure the handling of self modifying code. Furthermore, when optimisations such as concurrent code generation [7] or region-based code generation [28] are applied, these benchmarks will help to measure the effectiveness of these techniques.

Small blocks: This benchmark consists of short functions which tail call each other. To force code generation, the first word in each function is rewritten at the start of each iteration. This means that this benchmark also performs a large amount of indirect control flow, but the benchmark execution time should be dominated by code generation, except in extremely unusual cases (e.g. where code generation is extremely fast, or where the generated code is of extremely poor quality).

Large blocks: This benchmark has a single very large basic block with a repeated sequence of arithmetic instructions and where the first word of the block is rewritten at every execution. At the start of each iteration the inputs are read from a set of volatile variables, and the results are written back at the end of each iteration.

The code generation benchmarks need to be carefully written to stop the compiler from optimising away important

portions of the benchmark. For example, if the large block benchmark was naively written as a computation occurring on constant values, constant folding could be used to eliminate it. Similarly, each iteration of the benchmark involves a function call to the benchmark kernel. If that call can be inlined at compile time, then code generation will only occur once, at the first iteration of the benchmark, rather than once per iteration.

2) Control Flow Handling: Control flow handling in a simulator can be split into two groups: intra-page and inter-page. This split exists because intra-page control flow does not require a virtual address translation, as long as address mappings are not changed. Control flow can also be split into direct (where the branch target is known in advance, and is encoded as an absolute or relative path into the instruction) and indirect (where the branch target is read from memory or a register). A large amount of work has been done on optimising the various forms of control flow [28, 20, 15, 17].

SimBench includes benchmarks to test each of the four combinations of these types.

Inter Page Direct and Indirect These benchmarks consist of several short functions located on separate pages that tail call each other. In the Inter Page Indirect benchmark, the functions are called via difficult to predict function pointers, in order to defeat compiler optimisation.

Intra Page Direct and Indirect These benchmarks are similar to the Inter Page benchmarks, except that all the functions are on the same memory page.

In each of these benchmarks, we are careful to inhibit the compiler's ability to perform optimisations such as function inlining, in order to ensure that the benchmark operates correctly. Furthermore, defining the size of a 'page' is difficult since different architectures have different minimum page sizes. For example, the ARM architecture specifies a minimum page size of 1KB prior to ARMv6, while many other architectures (and new versions of ARM) have a minimum page size of 4KB. This does not significantly affect this benchmark at this time but would have to be revisited if SimBench were ported to an architecture with a minimum page size of more than 4KB. This is particularly relevant for the inter-page benchmarks since we need to ensure that the control flow used by these benchmarks does actually cross page boundaries.

3) Exception/Interrupt Handling: There is a large range of exception types in modern computer architectures. The most frequently encountered are virtual memory related exceptions and system calls. SimBench contains benchmarks to target these and other common exception and interrupt types.

Data Access Exception: Virtual memory related exceptions can be split into data and instruction exceptions. SimBench contains a benchmark to test each case. The data memory benchmark repeatedly attempts to access a memory location which is not mapped, generating an exception each time. The exception handler immediately returns to the next instruction after the memory access.

Instruction Access Exception: This benchmark repeatedly attempts to call a function located in a region of unmapped memory. Each call results in an exception, which is handled by returning to the next instruction after the function call (this requires some stack unwinding on architectures such as x86).

Undefined Instruction This benchmark uses an architecturally undefined instruction to trigger an exception. Most instruction sets support at least one such instruction (e.g., the `UD2` instruction in x86 [16], and the Architecturally Undefined Instruction Space in the ARM architecture [2]). The use of undefined instructions varies by architecture and application, but undefined instruction handling mechanisms can be used to perform emulation of floating point instructions in CPUs that do not have a floating point unit, or to provide backwards compatibility for legacy instructions or operations.

System Call: This benchmark attempts to measure the performance of performing a system call (sometimes known as a 'software interrupt'). The exact nature of system calls is architecture specific, but typically involves executing a 'syscall' instruction which generates an exception. This benchmark repeatedly executes such an instruction, with the exception handler returning to the next instruction.

External Software Interrupt: While the syscall benchmark uses a system call instruction to generate an exception, the interrupt controllers in most systems also support software generated interrupts. This benchmark mainly exercises the interrupt handling performance of the system under test.

4) I/O Infrastructure: Many modern platforms include devices that are manipulated using memory mapped registers. For example, communication with a screen device uses normal memory store instructions. SimBench tests memory mapped I/O in a fairly limited fashion – a platform-specific device is repeatedly accessed (preferably a device with no side effects and very limited processing required to evaluate).

Coprocessor accesses can also be used to communicate with a limited range of devices external to the CPU. For example, the ARM VFP extensions are encoded as coprocessor access instructions, and a lot of ARM system configuration is performed via a system control coprocessor.

Note that we are not seeking to benchmark any particular I/O operation, but rather to measure the base cost of *any* operation. By accessing a straightforward side-effect-free register, we are able to investigate the cost of performing an I/O access, without measuring a particular subsystem. For example, if we perform a complex operation through a coprocessor, then we are measuring the costs of performing that complex operation, rather than the costs associated with coprocessor accesses.

However, we must also be careful to select I/O and coprocessor accesses that cannot be trivially optimised away by the simulator. For example, a CPUID register read could potentially be optimised into a constant value by a sufficiently smart simulator (this is more difficult for memory mapped devices). Determining a 'safe' register to access is something that must be done separately for each architecture.

Memory Mapped Device Access: This benchmark repeatedly accesses a platform specific 'safe' device, using a memory access operation. For example, this might toggle an LED, or repeatedly read from a device ID register.

Coprocessor Access: This benchmark repeatedly accesses an architecture specific 'safe' coprocessor, using an architecture specific method. In the case of ARM, the Domain Access Control register is read from. In the case of x86, the mathematic coprocessor is repeatedly reset.

5) Memory System: The performance of the memory system is very important to the overall performance of a simulator. For example, in the SPEC benchmark suite compiled for the x86 architecture, up to 57% of instructions involve data memory accesses [6]. In a full system simulation of a system with an MMU, each memory access must perform a virtual to physical address translation, and any privilege checks required. Many real systems include a TLB to accelerate these translations, and many simulators include a similar structure. SimBench includes benchmarks designed to exercise both the hot path (a TLB hit) and the cold path (a TLB miss).

Since a 'cold' memory access involves performing an MMU translation, this benchmark necessarily examines the architecture specific MMU implementation. This can have a significant impact on the overall runtime of the benchmark, since e.g. a single level translation (such as an ARM section or supersection translation) is more straightforward than a two-level translation (such as a coarse page translation). The mapping of pages is handled by the architecture support package rather than the benchmark.

Cold Memory Access: This benchmark reserves a large portion of memory, and performs one memory read at the top of each page of that region. For this reason, each access results in a TLB miss and a 'cold-path' memory access.

Hot Memory Access: In this benchmark the same memory page is loaded from and stored to repeatedly. Each iteration could potentially consist of only two instructions (a load and a store instruction), so the benchmark loop is manually unrolled. This benchmark tests the common case for memory accesses.

Nonprivileged Access: Similar to the 'hot' memory access benchmark, except that the normal memory access is replaced with a non-privileged memory access for architectures which support this kind of access (such as ARM). Such nonprivileged accesses can be used to safely copy data from an operating system kernel into user memory (in order to return data from system calls, for example).

Also important to the overall performance of the system is how TLB operations are performed in simulation. This situation is complicated by multiprocessing support built in to many modern virtual memory systems e.g., the ASID in the ARM virtual memory system and the PCID in x86. These are very much architecture specific features which are difficult to assess in a portable fashion. These might be handled in a future version of SimBench. Two benchmarks are currently in SimBench targeted at TLB operations.

TLB Eviction: This benchmark tests TLB eviction operations. This is similar to the 'cold' memory access benchmark, except that it evicts the accessed page of memory from the data TLB after each iteration.

TLB Flush: The same as the TLB Eviction benchmark, except that the entire data TLB is flushed after each iteration.

Machine	ODROID-XU3	HP z440
CPU	Exynos 5422	Xeon E5-1620 v3
CPU GHz	2.0 (A15) 1.4 (A7)	3.5 (3.6 Boost)
Memory	2GB	16GB
Compiler	gcc 4.8.2	gcc 5.3.1
OS Name	Ubuntu 14.04	Fedora 21
OS Kernel	3.10.53	4.1.13

Fig. 5: Details of the real ARM and x86 hardware platforms used during our experiments. Only the Cortex-A15 cores of the ODROID platform are used.

C. Porting SimBench

When developing the SimBench suite, one of the main challenges was ensuring that the desired behaviour was obtained without making the suite difficult to port. None of the benchmarks themselves contain any architecture specific assembly or platform specific code – all such operations are handled by architecture and platform support packages. Porting SimBench to a new architecture or platform requires only that the new support packages are written, rather than requiring that each benchmark be individually ported.

Since each benchmark is written in standards-compliant C, several benchmarks contain structures designed to deliberately defeat compiler optimisations, especially when measuring control flow handling. This can make the benchmark code somewhat more difficult to produce and to understand, but cannot be avoided while still maintaining portability.

Porting to a new platform is straightforward – for example, each ARM platform library is made up of around 200 lines of C code. To implement the current benchmark set, the platform library must primarily manage the serial connection to the host, provide information on the system's memory layout, and provide an interface for platform specific operations such as triggering external software interrupts. Porting to a new architecture is more complex – the ARM architecture library is 570 lines of C and 400 lines of assembly. This mainly deals with bringing the machine out of reset, and managing the MMU and caches. The architecture library must also provide several architecture-specific operations such as executing a syscall, an undefined instruction, coprocessor accesses, etc.

III. Evaluation

In this section we evaluate SimBench on several simulators and hardware platforms. We begin by describing the simulators and hardware platforms we used. We then discuss the capability of SimBench to explain well-know performance gaps in the simulation landscape. Finally, we perform a more detailed analysis of the results for the popular QEMU simulator.

A. Environments

We evaluated SimBench on a variety of simulated and real platforms. We used x86 and ARM for the host platforms, detailed in Figure 5. We select QEMU-DBT (QEMU which utilises Dynamic Binary Translation), QEMU-KVM (QEMU with hardware-assisted virtualization support), SimIt-ARM (which, in our configuration, does not use DBT) and Gem5

	QEMU-DBT	SimIt-ARM	Gem5	QEMU-KVM	Native
Execution Model	DBT	Fast Interpreter	Interpreter	Direct	Direct
Memory Access	Multi-level Page Cache	Single Level Cache	Modelled TLB	Direct	Direct
Code Generation	Block-based	None	None	None	None
Control Flow					
Inter-Page	Block Cache	Interpreted	Interpreted	Direct	Direct
Intra-Page	Block Chaining	Interpreted	Interpreted	Direct	Direct
Exception Modelling					
Interrupts	Block Boundaries	Insn. Boundaries	Insn. Boundaries	Via Emulation Layer	Direct
Synchronous Exceptions	Side Exit	Interpreted	Interpreted	Direct	Direct
Undefined Instruction	Translated	Interpreted	Interpreted	Hypercall	Direct

Fig. 4: Table showing examples of how certain features are implemented on different evaluated platforms. These features should be picked up by SimBench as differences in the runtime of specific benchmarks.

(non cycle accurate version) as our simulators. Overall results for each experiment can be seen in Figure 7. Many popular or well known simulators only support user-mode simulation, and so SimBench cannot be used with these simulators.

B. Analysis

In this section we evaluate the capability of SimBench to explain performance gaps rather than just identify them. We first explain two well-known performance gaps in the domain of simulation: we compare two classical simulation techniques, Dynamic Binary Translation (DBT) and Interpretation, and use SimBench to explain the performance gaps between them. Then, we use SimBench to identify and explain benefits and weaknesses of executing code natively, versus using hardware-assisted virtualization. Finally we will evaluate the QEMU simulator with SimBench and SPEC over 20 different versions.

1) DBT vs Interpretation: Generally we would expect QEMU to significantly outperform SimIt-ARM and Gem5, due to its use of Dynamic Binary Translation (DBT), and this is reflected in most of the SimBench benchmark runtimes.

Since QEMU performs DBT, we expect workloads containing a lot of new or self-modifying code to perform slowly (since new/modified code must be translated when it is first encountered). This is indeed shown by SimBench: the Code Generation benchmarks are executed much more quickly on SimIt-ARM (which in our case executes instructions via a fast interpreter) than on QEMU. Gem5 also performs poorly on these benchmarks, despite also using an interpreter. This is due to the Gem5 interpreter being much more detailed in nature than that of SimIt-ARM (as it is intended to be used for cycle-accurate simulation).

Another situation where SimIt-ARM outperforms QEMU is the Cold Memory Access benchmark. The MMU model of SimIt-ARM is simpler than that of QEMU, and takes less time to evaluate on each TLB miss. QEMU supports multiple ARM architecture versions, including many extensions and variants, making page table lookups quite complex. However, SimIt-ARM supports only the ARMv5 architecture.

We would also expect QEMU to outperform SimIt-ARM and Gem5 on the Control Flow benchmarks, due to QEMU's use of block chaining, block caching, etc. However, the performance difference between QEMU and SimIt-ARM is not as great as might be expected. These benchmarks are intended to stress translation lookup systems, so actually having translations is not necessarily an advantage. However, QEMU's use of block chaining allows it to obtain a good speedup against SimIt-ARM on the Intra-Page Direct benchmark.

To conclude, SimBench is able to identify reasons for the performance gap between DBT and interpretation-based simulation tools.

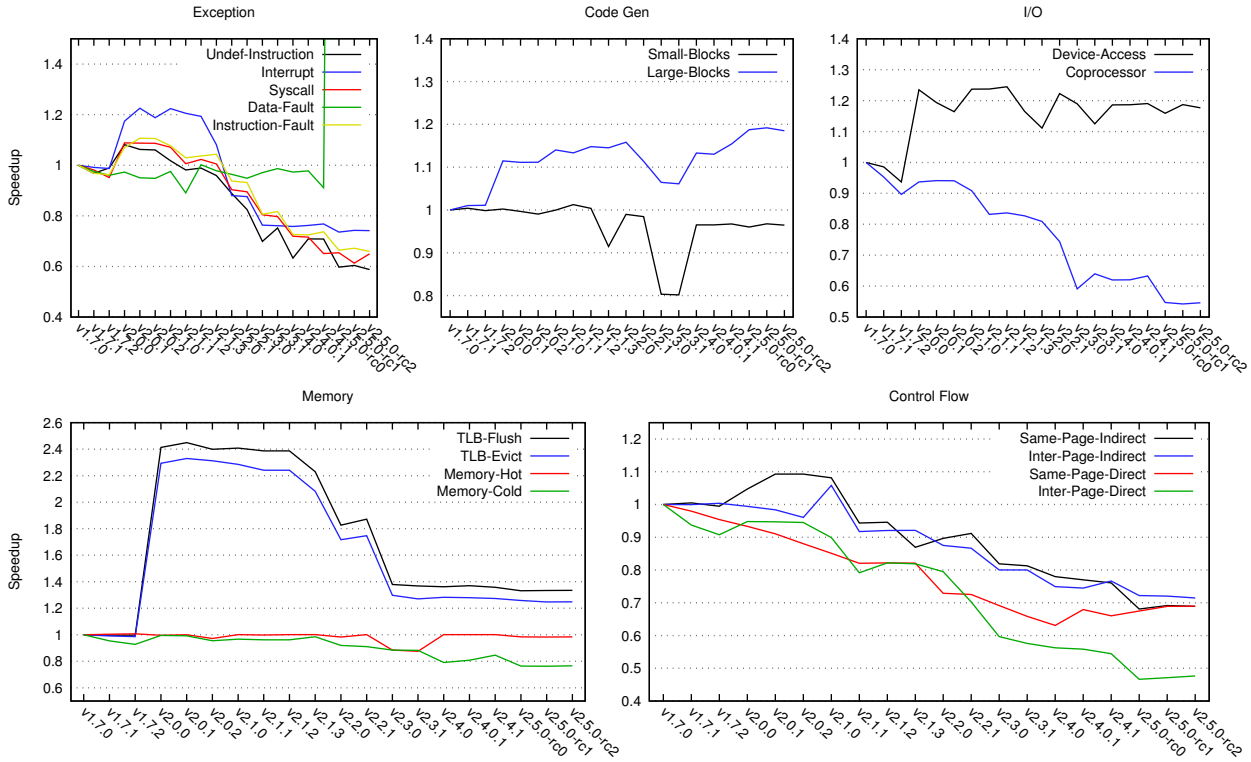
2) Virtualization against Native Performance: The performance of the SPEC benchmark using QEMU-KVM are comparable to those using native hardware on ARM and x86.

By using SimBench on ARM we note that the performance of QEMU-KVM and Native Hardware are fairly similar in most benchmarks except Control Flow, External Software Interrupt, and I/O. We believe that the poor performance of KVM in the Control Flow benchmarks is due to KVM being relatively unstable in our selected kernel version on the ODROID-XU3 platform. The External Software Interrupt and I/O benchmarks also run significantly faster on real hardware when compared against QEMU-KVM. Both of these benchmarks involve accesses to external devices. In a virtualized environment, accesses to emulated devices are trapped and handled by the virtualization infrastructure, which is much more costly than accessing actual hardware. On investigating the External Software Interrupt benchmark result for QEMU-KVM, we found that the result was particularly poor due to 'unsupported operation' messages being written to the system log (despite the correct behaviour being performed by KVM).

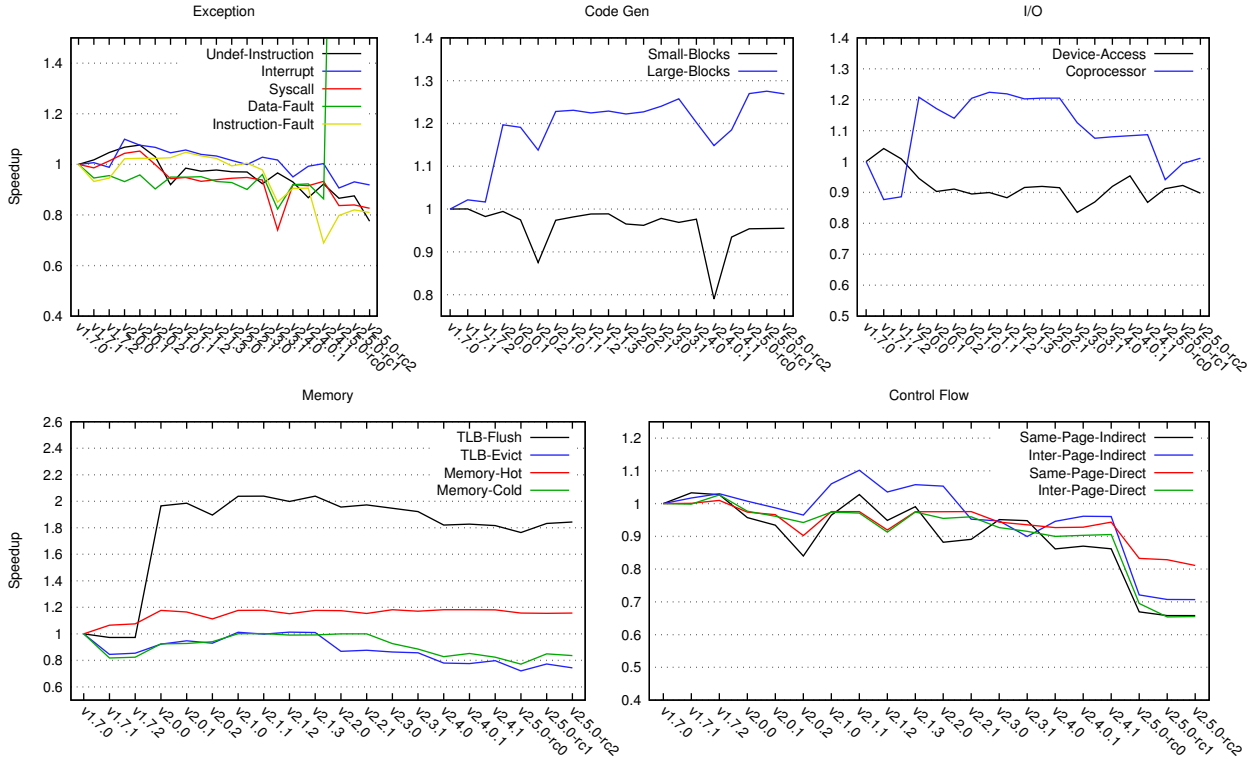
On x86, KVM and native hardware obtain similar results, with a few exceptions. First, in some cases the QEMU-KVM results are slightly faster. This is likely to be due to the use of Intel SpeedStep (frequency boosting) by the KVM host operating system, which is not enabled on the bare metal platform. Several benchmarks have fairly significant performance differences, particularly the Undefined Instruction, External Software Interrupt, and Memory Mapped Device benchmarks. These benchmarks all include operations which involve trapping into KVM, which has a performance overhead.

To conclude, SimBench successfully identified a well-known caveat of virtualization, when using application benchmarks would lead us to consider them as equivalent.

3) SPEC performance variation in QEMU: We ran both the SPEC2006 Integer suite and SimBench on each version of QEMU, going back several years, using an x86 host (outlined



(a) ARM Guest



(b) x86 Guest

Fig. 6: Graphs showing the performance of the QEMU-DBT simulator on each category of the SimBench benchmarks. For the Data-Fault Exception benchmark we observed a speedup of around 8x on ARM and 4x on x86 for the 2.5.0 versions of QEMU-DBT (result which is off the scale on the corresponding graphs). The host (an HP z440 workstation) is specified in Figure 5.

Benchmark	ARM Guest (seconds)					x86 Guest (seconds)		
	QEMU-DBT	SimIt-ARM	Gem5	QEMU-KVM	Hardware	QEMU-DBT	QEMU-KVM	Hardware
Code Generation								
Small Blocks	7.758	0.453	18.667	4.377	1.914	6.201	1.533	0.029
Large Blocks	10.609	0.778	47.612	2.048	0.560	9.222	0.319	0.068
Control Flow								
Inter-Page Direct	11.766	14.388	656.301	124.209	32.229	7.601	0.877	0.966
Inter-Page Indirect	10.134	12.885	495.388	138.231	61.367	8.643	0.419	0.492
Intra-Page Direct	11.320	70.440	2478.621	823.059	52.005	9.898	2.091	2.236
Intra-Page Indirect	9.476	10.038	394.483	104.974	29.505	6.907	0.335	0.336
Exception Handling								
Data Access Exception	7.290	11.499	114.562	6.023	6.286	16.975	6.503	6.219
Insn. Access Exception	8.114	23.444	218.416	7.372	9.062	11.094	7.679	7.120
Undefined Instruction	9.286	22.393	207.291	3.368	6.702	13.691	54.549	8.629
System Call	8.621	23.657	276.409	3.494	6.952	11.898	8.063	7.898
Ext. Software Interrupt	10.584	42.094	–†	2817.653	21.670	14.414	112.843	6.571
I/O								
Memory Mapped Device	12.777	16.681	–†	4407.052	16.502	29.358	316.737	0.112
Coprocessor Access	8.166	9.883	441.189	96.406	6.001	17.469	438.045	493.731
Memory System								
Hot Memory Access	9.879	90.705	4886.551	17.504	35.006	10.422	7.833	7.826
Cold Memory Access	9.844	3.222	117.099	12.303	6.943	13.788	0.558	0.518
Nonprivileged Access	10.227	123.741	824.847	8.067	1.051	–	–	–
TLB Eviction	13.998	1.777	29.093	1.952	1.150	1.511	0.666	0.602
TLB Flush	12.728	1.572	26.854	4.891	3.069	11.120	1.030	0.582

Fig. 7: Results from running SimBench on a variety of platforms. The target machines are outlined in Figure 5. QEMU Version 2.5.0-rc2 was used for all experiments involving QEMU. † The functionality is not implemented in the Gem5 simulator.

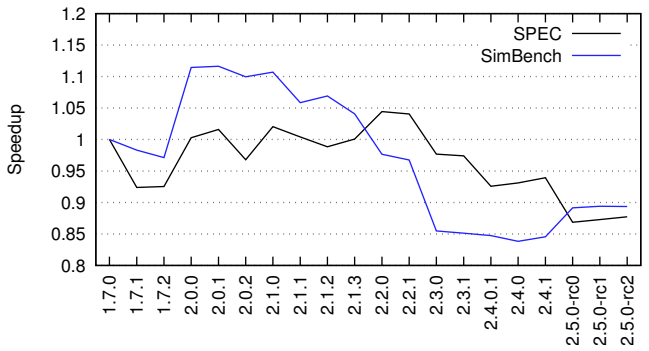


Fig. 8: Graph of geometric mean of speedup of QEMU (baseline is v1.7.0) running SPEC2006 Integer benchmarks, and SimBench, across a range of QEMU versions.

in Figure 5) and an ARM Linux host. As can be seen in Figure 8, QEMU is generally decreasing in performance with each released version. This is reflected in both SPEC and SimBench results. Although the performance of SimBench does not precisely reflect that of SPEC (i.e., you could not accurately use one to predict the other), some general trends and interesting features can be observed.

Firstly, version 2.0.0 of QEMU provides a large improvement in most SimBench categories, as well as an improvement in SPEC performance. The QEMU v2.0 Change Log [1] notes that this version includes "Improvements to the TCG optimiser" which may explain this performance improvement.

Version 2.5.0-rc0 sees a significant improvement in data fault handling performance (as can be seen in Figure 6). However, this is not matched by any improvement in SPEC performance. Such data access faults are uncommon so it would not be expected that an improvement in data fault handling performance would produce a significant improvement in overall application performance. However, data fault handling performance might be more important when dealing with memory constrained systems where pages of data are frequently swapped out to disk.

Finally, a degradation in performance on control flow benchmarks can be observed. This is perhaps the most problematic issue revealed by SimBench, since control flow handling is fundamental to simulation. Although some of the control flow benchmarks also contain computation (in order to defeat compiler optimisation), this does not account for the large degradation observed. Exception handling also shows a significant and consistent reduction in performance (except for data fault exception handling). Although this makes up a smaller component of overall simulation performance (as shown by the operation densities shown in Figure 3), it may become significant if performance in this area continues to degrade.

IV. Related Work

While there has been no prior work on benchmarks specifically aimed at full-system simulators, numerous benchmark suites and methodologies targeting applications, kernels and virtualization have been developed. We review the related work on these benchmarking approaches and briefly summarise current practice of simulator benchmarking.

A. Application and kernel benchmarks

Probably the most widely used application benchmark is the SPEC [14] suite, which contains modified versions of existing large programs such as Perl and gcc, along with several data

sets. SPEC is typically used for evaluating improvements in computer architecture and compiler technologies. Other benchmark suites include EEMBC [21], which contains a large suite of kernels representing a range of embedded applications, PARSEC [4], which is designed to measure the performance of CMP systems and PARBOIL [29] which contains a number of throughput-computing benchmarks suitable for evaluating e.g. GPU performance. There are a large number of other benchmark suites, and a general survey on performance evaluation can be found in e.g. [18].

B. Virtualization benchmarks

The Kernel-based Virtual Machine (KVM) [13] is a virtualization infrastructure for the Linux kernel that turns it into a hypervisor. It requires a processor with hardware virtualization extensions and has been ported to a variety of architectures, including ARM [12].

VMmark [30] is virtual machine benchmark suite, which measures the performance of virtualized servers while running under load on physical hardware. VITS [34] is a micro-benchmark suite designed to measure cache, memory bandwidth, CPU, network and disk performance in virtualized environments. Whilst these categories broadly match our SimBench benchmarks, the overheads for virtualization and simulation are quite different and demand specialised tests. For example, Dynamic Binary Translation is usually not used in virtualization systems (which instead use hardware extensions to provide virtualization). Similarly, the host system MMU can be used in a virtualized context, whereas in a simulator, memory accesses must usually be completely emulated.

C. DBT and simulator benchmarking

The main sources of DBT overhead are characterised in [9]. Furthermore, operations and associated overheads are classified into five categories, and their contribution to the overall overhead quantified. SimBench builds on this work and provides measurements for each of the five categories and beyond. Generally, simulators are benchmarked using standard benchmark suites, such as those described above. The integer portion of SPEC2006 is popular (used in e.g. [32, 24]). Targeted microbenchmarks are occasionally provided [19] although precise details of these benchmarks are not always available. The EEMBC suite has been used for simulator evaluation in e.g. [31]. SimIt-ARM [26] is benchmarked against a mix of applications from Media Bench and SPEC INT 2000. [27] considers performance changes of simulator benchmarks when context switches are incorporated.

V. Conclusion

In this paper we have presented a methodology for the systematic performance evaluation of full-system instruction set simulators. Rather than relying on application benchmarks, which only measure a fraction of performance-critical features we use a set of micro-benchmarks specifically aimed at those operations, which are costly to implement in simulation.

We have applied SimBench to QEMU, targeting the ARM and x86 architectures, Gem5 and SimIt-ARM. In addition, we have performed additional evaluations against QEMU-KVM virtualized platform as well as native execution. We show that the ARM port of QEMU suffers from a continuous performance degradation issue, which we are able to pinpoint to central simulation operations impossible to detect using e.g. SPEC CPU2006 benchmarks. Future work includes the development of additional targeted benchmarks as well as the application of SimBench to full-system DBT virtualization systems like STAR [25]. We might also investigate the use of SimBench-like kernels for sandbox detection.

VI. Availability

SimBench is available at https://bitbucket.org/simbench/simbench, under a New BSD license.

VII. Acknowledgements

We acknowledge funding by the EPSRC grant PAMELA EP/K008730/1.

References

[1] QEMU Version 2.0 Change Log.

[2] *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition*. ARM Holdings plc, 2011.

[3] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX*, ATEC '05. USENIX Association, 2005.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, 2008.

[5] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The Gem5 Simulator. *SIGARCH Comput. Archit. News*, Aug. 2011.

[6] S. Bird, A. Phansalkar, L. K. John, A. Mercas, and R. Idukuru. Performance characterization of SPEC CPU benchmarks on Intel's Core microarchitecture based processor. In *SPEC Benchmark Workshop*, Jan. 2007.

[7] I. Böhm, T. J. Edler von Koch, S. C. Kyle, B. Franke, and N. Topham. Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, 2011.

[8] I. Böhm, B. Franke, and N. Topham. Cycle-accurate performance modelling in an ultra-fast just-in-time dynamic binary translation instruction set simulator. In *Transactions on High-Performance Embedded Architectures and Compilers (HiPEAC 11)*, 2011.

[9] E. Borin and Y. Wu. Characterization of DBT overhead. In *IEEE International Symposium on Workload Characterization (IISWC 2009)*, 2009.

[10] F. Brandner. Precise simulation of interrupts using a roll-back mechanism. In *Proceedings of th 12th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '09, 2009.

[11] J. Cong, Z. Fang, M. Gill, and G. Reinman. PARADE: A cycle-accurate full-system simulation platform for accelerator-rich architectural design and exploration. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '15, 2015.

[12] C. Dall and J. Nieh. KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, 2014.

[13] I. Habib. Virtualization with KVM. *Linux Journal*, Feb. 2008.

[14] J. L. Henning. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News*, Sept. 2006.

[15] J. D. Hiser, D. Williams, W. Hu, J. W. Davidson, J. Mars, and B. R. Childers. Evaluating indirect branch handling mechanisms in software dynamic translation systems. In *International Symposium on Code Generation and Optimization (CGO '07)*, March 2007.

[16] *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, Santa Clara, CA, USA, February 2014.

[17] N. Jia, C. Yang, J. Wang, D. Tong, and K. Wang. SPIRE: Improving dynamic binary translation through SPC-indexed indirect branch redirecting. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '13, 2013.

[18] L. K. John. 8.2 performance evaluation: Techniques, tools, and benchmarks. *The Computer Engineering Handbook*, 2002.

[19] J. Jovic, S. Yakoushkin, L. Murillo, J. Eusse, R. Leupers, and G. Ascheid. Hybrid simulation for extensible processor cores. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '12, 2012.

[20] T. Koju, X. Tong, A. I. Sheikh, M. Ohara, and T. Nakatani. Optimizing indirect branches in a system-level dynamic binary translator. In *Proceedings of the 5th Annual International Systems and Storage Conference*, SYSTOR '12, 2012.

[21] M. Levy. EEMBC and the Purposes of Embedded Processor Benchmarking. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2005, March 20-22, 2005, Austin, Texas, USA, Proceedings*. IEEE Computer Society, 2005.

[22] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: a full system simulation platform. *j-COMPUTER*, Feb. 2002.

[23] L. McVoy and C. Staelin. Lmbench: Portable tools for performance analysis. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ATEC '96, pages 23–23, Berkeley, CA, USA, 1996. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1268299.1268322.

[24] A. A. Nair and L. K. John. Simulation points for SPEC CPU 2006. In *IEEE International Conference on Computer Design (ICCD 2008)*, Oct 2008.

[25] N. Penneman, D. Kudinskas, A. Rawsthorne, B. D. Sutter, and K. D. Bosschere. Evaluation of dynamic binary translation techniques for full system virtualisation on ARMv7-A. *Journal of Systems Architecture*, 65:30–45, 2016.

[26] W. Qin, J. D'Errico, and X. Zhu. A Multiprocessing Approach to Accelerate Retargetable and Portable Dynamic-compiled Instruction-set Simulation. In *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '06, 2006.

[27] R. S. Shindi and S. Cooper. Evaluate the performance changes of processor simulator benchmarks when context switches are incorporated. In *Proceedings of the 2006 Annual ACM SIGAda International Conference on Ada*, SIGAda '06, 2006.

[28] T. Spink, H. Wagstaff, B. Franke, and N. Topham. Efficient code generation in a region-based dynamic binary translator. In *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES '14, New York, NY, USA, 2014.

[29] J. A. Stratton, C. Rodrigues, I. J. Sung, N. Obeid, L. W. Chang, N. Anssari, D. Liu, G., and W. W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Technical report, University of Illinois at Urbana-Champaign, 2012.

[30] VMWare Inc. VMmark Virtualization Benchmarks. http://www.vmware.com/uk/products/vmmark, 2015.

[31] H. Wagstaff, M. Gould, B. Franke, and N. Topham. Early Partial Evaluation in a JIT-compiled, Retargetable Instruction Set Simulator Generated from a High-level Architecture Description. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, 2013.

[32] Z. Wang, J. Li, C. Wu, D. Yang, Z. Wang, W.-C. Hsu, B. Li, and Y. Guan. HSPT: Practical implementation and efficient management of embedded shadow page tables for cross-ISA system virtual machines. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '15, 2015.

[33] E. Witchel and M. Rosenblum. Embra: Fast and flexible machine simulation. In *Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '96, 1996.

[34] P. Yuan, C. Ding, L. Cheng, S. Li, H. Jin, and W. Cao. VITS test suite: A micro-benchmark for evaluating performance isolation of virtualization systems. In *International Conference on e-Business Engineering (ICEBE)*, Nov 2010.